COMPSCI 389
Introduction to Machine Learning

Models, Algorithm Template, Nearest Neighbor

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

Review

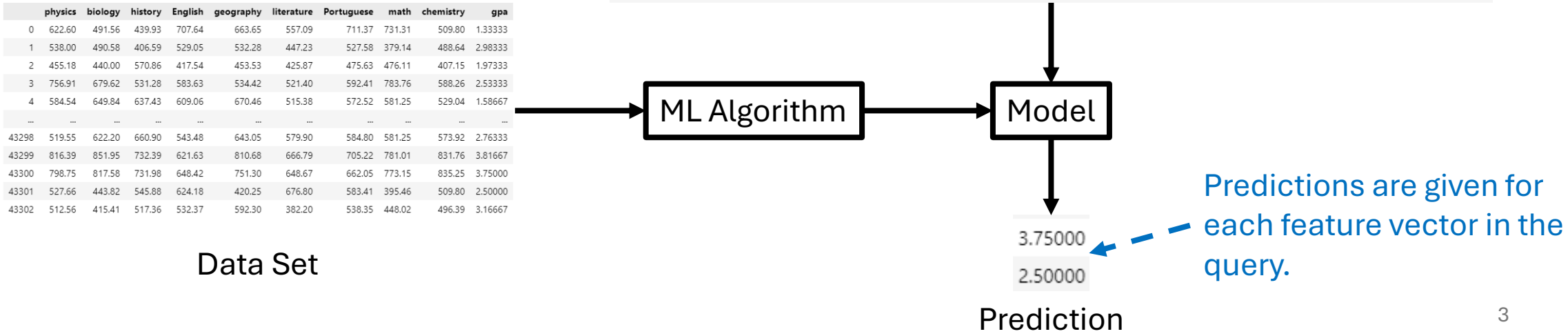
- Input output pair (X, Y)
  - X: Input, features, attributes, covariates, or predictors
    - Numerical (discrete/continuous), categorical (nominal/ordinal), etc.
  - Y: Output, label, or target
    - Regression: Y is continuous.
    - Classification: Y is discrete
- Data set: $(X_i, Y_i)_{i=1}^n$
- Query: An additional input X
- Goal: Predict the label Y associated with X.

Models (Supervised Learning)

- A **model** is a mechanism that maps input data to predictions.
- (Offline) **ML algorithms** take data sets as input and produce models as output.

Online ML algorithms can receive data over time, improving their models as more data becomes available.

A query can be one **or more** feature vectors.



	physics	biology	history	English	geography	literature	Portuguese	math	chemistry	gpa
0	622.60	491.56	439.93	707.64	663.65	557.09	711.37	731.31	509.80	1.33333
1	538.00	490.58	406.59	529.05	532.28	447.23	527.58	379.14	488.64	2.98333
2	455.18	440.00	570.86	417.54	453.53	425.87	475.63	476.11	407.15	1.97333
3	756.91	679.62	531.28	583.63	534.42	521.40	592.41	783.76	588.26	2.53333
4	584.54	649.84	637.43	609.06	670.46	515.38	572.52	581.25	529.04	1.58667
...	...	...	...	...	...	...	...	...	...	...
43298	519.55	622.20	660.90	543.48	643.05	579.90	584.80	581.25	573.92	2.76333
43299	816.39	851.95	732.39	621.63	810.68	666.79	705.22	781.01	831.76	3.81667
43300	798.75	817.58	731.98	648.42	751.30	648.67	662.05	773.15	835.25	3.75000
43301	527.66	443.82	545.88	624.18	420.25	676.80	583.41	395.46	509.80	2.50000
43302	512.56	415.41	517.36	532.37	592.30	382.20	538.35	448.02	496.39	3.16667

Data Set

Training/Fitting

- ML algorithms could take a data set and query at the same time and output a prediction for the query.
  - Each time a new query is given, the algorithm re-processes the entire data set.
- **Idea**: More efficient to preprocess the data set, computing relevant statistics and quantities.
  - Given a query, the algorithm might reference the statistics and quantities it computed without re-referencing the data set at all!
  - This pre-processing of the data set is called **training.**
    - Sometimes: “Training the model”
    - Sometimes: “Fitting the model to data”
    - Sometimes: “Pre-processing data”

Scikit-Learn Models

- Scikit-Learn is a popular ML library in python.
- It has objects called “models”.
  - These “models” are more than just models – they are complete ML algorithms.

Scikit-Learn Models

- Scikit-Learn models implement the functions:
  - **`fit(self, X, y)`**: The function for fitting the model to the data (training the model given the data / preprocessing the data).
    - `X`: A 2D array-like structure (e.g., DataFrame) representing the features. Each row is a point and each column is a feature.
    - `y`: A 1D array-like structure (e.g., Series) representing the target values
    - Returns `self` to simplify chaining together operations.
  - **`predict(self, X)`**: The function for producing predictions given queries.
    - `X`: A 2D array-like structure representing the data for which predictions are to be made. Each row is a sample and each column is a feature.
    - Returns a numpy array of predicted labels/values.
- **Note**: Ideally fit and predict are compatible with `X` and `y` being DataFrames or numpy arrays.

Scikit-Learn Models

```
from sklearn.base import BaseEstimator
import numpy as np

class CustomMLAlgorithm(BaseEstimator):
    def __init__(self, param1=1, param2=2):
        # Initialization code
        self.param1 = param1
        self.param2 = param2
```

Scikit-Learn Models

```
from sklearn.base import BaseEstimator
import numpy as np

class CustomMLAlgorithm(BaseEstimator):
    def __init__(self, param1=1, param2=2):
        # Initialization code
        self.param1 = param1
        self.param2 = param2

    def fit(self, X, y):
        # Training code
        # Implement your training algorithm here
        return self
```

Scikit-Learn Models

```
from sklearn.base import BaseEstimator
import numpy as np

class CustomMLAlgorithm(BaseEstimator):
    def __init__(self, param1=1, param2=2):
        # Initialization code
        self.param1 = param1
        self.param2 = param2

    def fit(self, X, y):
        # Training code
        # Implement your training algorithm here
        return self

    def predict(self, X):
        # Prediction code
        # Implement your prediction algorithm here
        return np.zeros(len(X))
```

Scikit-Learn Models

```
from sklearn.base import BaseEstimator
import numpy as np

class CustomMLAlgorithm(BaseEstimator):
    def __init__(self, param1=1, param2=2):
        # Initialization code
        self.param1 = param1
        self.param2 = param2

    def fit(self, X, y):
        # Training code
        # Implement your training algorithm here
        return self

    def predict(self, X):
        # Prediction code
        # Implement your prediction algorithm here
        return np.zeros(len(X))
```

- Given data set (X,y) and query:

```
model = CustomMLAlgorithm()
model.fit(X,y)
predictions = model.predict(query)
```

Nearest Neighbor

- A particularly simple yet effective ML algorithm based on the core idea:

  When presented with a query, find the data point (row) that is most similar to the query and give the label associated with this most-similar point as the prediction.

- We can map this to fit/predict functions:
  - `fit`: Store the data
  - `predict`: For each query row do the following
    - Loop over each row in the training data, computing the Euclidean distance between the query and the row.
    - Create an array holding the labels from the rows with the smallest distance to the query feature vector (often just one element).
    - Return an arbitrary (e.g., random) element of the array.

Query:												
414.7	456.95	705.58	499.43	513.53	543.15	408.51	384.38	442.49				
physics	biology	history	English	geograp	literatu	Portugu	math	chemistry		distance		gpa
622.6	491.56	439.93	707.64	663.65	557.09	711.37	731.31	509.8				1.33333
538	490.58	406.59	529.05	532.28	447.23	527.58	379.14	488.64	Nearest Neighbor			2.98333
455.18	440	570.86	417.54	453.53	425.87	475.63	476.11	407.15	→		Prediction →	1.97333
756.91	679.62	531.28	583.63	534.42	521.4	592.41	783.76	588.26				2.53333
584.54	649.84	637.43	609.06	670.46	515.38	572.52	581.25	529.04				1.58667
325.99	466.74	597.06	554.43	535.77	717.03	477.6	503.82	422.92				1.66667
622.6	587.04	598.85	603.32	690.7	652.86	533.05	755.3	628.73				3.72333
527.65	559.99	758.37	669.71	645.62	648.67	539.23	470.78	486.13				3.08333
647.64	687.83	630.61	613.95	557.43	739.94	557.27	557.14	632.54				0

Implementing Nearest Neighbor

- See `5 Nearest Neighbor.ipynb`.
- When going through the following slides, you can follow along in this notebook.

`__init__`

- Our initial nearest neighbor implementation has no hyperparameters or initialization to perform, so we do not implement `__init__`.
  - **Hyperparameter**: A setting or value that changes the behavior of an ML algorithm.
  - We will revisit hyperparameters later when we encounter them.

fit

- We want to be compatible with DataFrames or NumPy arrays.
- We will convert DataFrames to NumPy arrays.

```python
class NaiveNearestNeighbor(BaseEstimator):
    def fit(self, X, y):
        # Convert X and y to NumPy arrays if they are DataFrames.
        # This makes fit compatible with numpy arrays or DataFrames
        if isinstance(X, pd.DataFrame):
            X = X.values
        if isinstance(y, pd.Series):
            y = y.values

        # Store the training data and labels.
        self.X_data = X
        self.y_data = y
        return self
```

predict

- Notice that predict returns a prediction for each row in X.

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # We will iteratively load predictions, so it starts empty
    predictions = []

    # Loop over rows in the query
    for x in X:
        # Compute distances from x to all points in X_data.
```

```
        # Find the nearest neighbors (handling ties)
```

```
        # Append this label to predictions
```

```
    # Return the array of predictions we have created
    return np.array(predictions)
```

predict

- Notice that predict returns a prediction for each row in X.

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # We will iteratively load predictions, so it starts empty
    predictions = []

    # Loop over rows in the query
    for x in X:
        # Compute distances from x to all points in X_data.
        # differences = self.X_data - x
        # squared_differences = differences ** 2
        # sum_squared_differences = np.sum(squared_differences, axis=1)
        # distances = np.sqrt(sum_squared_differences)
        distances = np.sqrt(np.sum((self.X_data - x) ** 2, axis=1))

        # Find the nearest neighbors (handling ties)



        # Append this label to predictions


    # Return the array of predictions we have created
    return np.array(predictions)
```

predict

- `np.where` returns a tuple of arrays, one for each dimension.
  - If we gave a 2d array, [0] would be the row index and [1] would be the col index.
  - distances is 1-D, so there is only one element in the tuple here, so we pass [0] to get the element
- If many equally close, this implementation selects the first one.

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # We will iteratively load predictions, so it starts empty
    predictions = []

    # Loop over rows in the query
    for x in X:
        # Compute distances from x to all points in X_data.
        # differences = self.X_data - x
        # squared_differences = differences ** 2
        # sum_squared_differences = np.sum(squared_differences, axis=1)
        # distances = np.sqrt(sum_squared_differences)
        distances = np.sqrt(np.sum((self.X_data - x) ** 2, axis=1))

        # Find the nearest neighbors (handling ties)
        min_distance = np.min(distances)
        nearest_neighbors = np.where(distances == min_distance)[0]
        nearest_label = self.y_data[nearest_neighbors[0]]

        # Append this label to predictions


    # Return the array of predictions we have created
    return np.array(predictions)
```

predict

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # We will iteratively load predictions, so it starts empty
    predictions = []

    # Loop over rows in the query
    for x in X:
        # Compute distances from x to all points in X_data.
        # differences = self.X_data - x
        # squared_differences = differences ** 2
        # sum_squared_differences = np.sum(squared_differences, axis=1)
        # distances = np.sqrt(sum_squared_differences)
        distances = np.sqrt(np.sum((self.X_data - x) ** 2, axis=1))

        # Find the nearest neighbors (handling ties)
        min_distance = np.min(distances)
        nearest_neighbors = np.where(distances == min_distance)[0]
        nearest_label = self.y_data[nearest_neighbors[0]]

        # Append this label to predictions
        predictions.append(nearest_label)

    # Return the array of predictions we have created
    return np.array(predictions)
```

Applying `NaiveNearestNeighbor` to GPA Data

```
df = pd.read_csv("https://people.cs.umass.edu/~pthomas/courses/COMPSCI_389_Spring2024/GPA.csv", delimiter=',')
# Split the inputs from the outputs
X = df.iloc[:,:-1]
y = df.iloc[:,-1]
```

Applying `NaiveNearestNeighbor` to GPA Data

```
# Create the Nearest Neighbor Model
model = NaiveNearestNeighbor()

# Call fit to train the model (in this case, just store the data set)
model.fit(X,y)

# Create two query points (in reality these would be new applicants)
query = X.head(2)

# Get predictions for the query points
predictions = model.predict(query)
print(predictions)
```

✓ 0.0s

```
[1.33333 2.98333]
```

Optimizing Nearest Neighbor Search

- Our `predict` function loops over the entire data set for each query point
- We can use data structures for finding nearest neighbors to make our implementation more efficient.
  - **K-D Trees**: Effective for low-dimensional data, but performance decreases with higher dimensions.
  - **Ball Trees**: Better suited for higher dimensional spaces.
- SciKit-Learn includes optimized implementations of both
- We will update our implementation to use a K-D tree.

```python
from sklearn.neighbors import KDTree
```

```python
class NearestNeighbor(BaseEstimator):
    def fit(self, X, y):
        # Convert X and y to NumPy arrays if they are DataFrames.
        # This makes fit compatible with numpy arrays or DataFrames
        if isinstance(X, pd.DataFrame):
            X = X.values
        if isinstance(y, pd.Series):
            y = y.values

        # Store the training data and labels.
        self.X_data = X
        self.y_data = y

        # Create a KDTree for efficient nearest neighbor search
        self.tree = KDTree(X)

        return self
```

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # Query the tree for the nearest neighbors of all points in X.
    # ind will be a 2D array where ind[i,j] is the index of the
    # j'th nearest point to the i'th row in X.
    dist, ind = self.tree.query(X, k=1)

    # Extract the nearest labels.
    # ind[:,0] are the indices of the nearest neighbors to each
    # query (each row in x))
    return self.y_data[ind[:,0]]
```

`k` specifies the number of nearest neighbors to query.

`dist` is a 2D array holding the distances to each of the nearest neighbors of each query.

`y_data` was stored during the `fit` call.

```
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # Query the tree for the nearest neighbors of all points in X.
    # ind will be a 2D array where ind[i,j] is the index of the
    # j'th nearest point to the i'th row in X.
    dist, ind = self.tree.query(X, k=1)

    # Extract the nearest labels.
    # ind[:,0] are the indices of the nearest neighbors to each
    # query (each row in x))
    return self.y_data[ind[:,0]]
```

Implementation Comparison

- **Note**: When there are multiple nearest neighbors, the algorithms may not return the same values.
- Let's run each implementation of Nearest Neighbor 100 times on the GPA data (with 2 queries each time).
  - Which will be faster?
  - How much faster?

```
Average runtime for NaiveNearestNeighbor: 0.003327179999905638 3 seconds
Average runtime for NearestNeighbor: 0.0805909420002717 5 seconds
```

- **Question**: Why do you think our naïve algorithm was faster?
- **Answer**: The overhead cost of building the K-D tree didn't pay of with just 2 queries.

Implementation Comparison (cont.)

- Let’s run 5,000 points through as queries.
- Recall: 43,303 points total.

```
Average runtime for NaiveNearestNeighbor: 8.219239899946842 seconds
Average runtime for NearestNeighbor: 0.09280269994633272 seconds
```

Intermission

- Class will resume in 5 minutes.
- Feel free to:
  - Stand up and stretch.
  - Leave the room.
  - Talk to those around you.
  - **Write a question on a notecard and add it to the stack at the front of the room.**